MAIA: An Event-based Modular Architecture for Intelligent Agents

J. Fernando Sánchez-Rada
Carlos A. Iglesias
and Miguel Coronado
Grupo de Sistemas Inteligentes
Universidad Politécnica de Madrid
Email: {jfernando, cif, miguelcb}@dit.upm.es

***Abstract*—Online services are no longer isolated. The release of public APIs and technologies such as web hooks are allowing users and developers to access their information easily. Intelligent agents could use this information to provide a better user experience across services, connecting services with smart automatic behaviours or actions. However, agent platforms are not prepared to easily add external sources such as web services, which hinders the usage of agents in the so-called Evented or Live Web. As a solution, this paper introduces an event-based architecture for agent systems, in accordance with the new tendencies in web programming. In particular, it is focused on personal agents that interact with several web services. With this architecture, called MAIA, connecting to new web services does not involve any modification in the platform.**

***Keywords*—Agent architecture, evented web, events, web hooks, jason**

I. Introduction

Agent architectures provide a valuable general guideline for designing and implementing agent applications [1] and have been a very active research topic in the agent community. In the 1990s, research interest was focused on the investigation of architectural issues raised by three influential threads of agent research (i.e. reactive agents, deliberative agents and interacting agents), as collects the excellent survey by Müller [2].

Software agent platforms are usually specialized in a particular agent architecture.For instance, most platforms for deliberative agents have adopted the Belief-Desire-Intention (BDI) model, as Jadex [3], Jack [4] or Jason [5], while the most popular agent platform for interacting agents, Jade [6], is based on FIPA [7]. Some of these platforms provide facilities to combine reasoning and interacting features, such as Jadex or Jason, which can be integrated with Jade.

The BDI architecture defined by Rao and Georgeff [8] is based on the original model proposed by Bratman for modelling human reasoning [9]. The BDI abstract architecture models human-like reasoning by capturing the mentalistic notions of belief, desire and intention, which are processed according to a generic interpreter. This interpreter assumes that events are atomic and recognized after they have occurred.

Traditionally, both messages and percepts have been managed in the same interpretation cycle, since both are considered forms of external events. As a result, most agent implementations mix reasoning processes with the communication logic and make them hard to reuse, debug and develop. Recently, several works such as ACRE [10] and Alfonso et al. [11] have proposed to delegate conversation management in a specific module external to the agent reasoning process. The interaction between these two modules is done through actions and perceptions. The reasoning module can reason about the outcomes of every conversation through a set of predefined perceptions, and then execute several actions to manage the status of those conversations (e.g. cancelling, forgetting or retrying a conversation).

Furthermore, agent platforms do not provide standardised mechanisms to integrate sensory information. This integration of sensors and actuators typically requires extending the basic agent architecture and a deep understanding of its implementation.

On the other hand, gathering information from external sources is a key aspect of any agent system. Lately, we are relying more and more on web services to store, share and generate new information.

Several works have proposed different mechanisms for integrating agents and web services, as surveyed in [12]. The existing solutions provide mappings between addressing and messaging schemes in web services and agent systems, and are implemented using a gateway that publishes web service descriptions into FIPA's directory facilitator and vice-versa. Nevertheless, there are application domains such as personal agents where the FIPA platform infrastructure is not needed but there is still the need to invoke services as a standard action.

A new trend in web service development is relying on event based interaction to allow services to interact. So much so that it is leading to a new generation of the web, called *real time web* or *evented web* [13]. This new wave of web services is characterised by its capability to process incoming events originated by a wide range of sources, such as social networks, service notifications or sensors.

Our proposal consists in overcoming the typical limitations in agent architectures while keeping them up to the current scenario. We do so by providing an event-based perspective to the internal composition of agent modules. This paper also explains how this architecture, called event-based Modular Architecture for Intelligent Agents (MAIA), can be used in applications that interact with a variable and increasing number of services, as well as its inner workings and implementation

challenges. To illustrate this, we also present an implementation of a personal cloud agent using MAIA.

This paper is structured as follows: Section III presents an overview of the architecture and describes its components in detail; Section IV covers the format and purpose of events; Section V shows how to use MAIA to build a personal agent; Section VI goes through related work; and in Section VII we present our conclusions and future work.

II. Event-based programming

Event-based programming [14], also called Event-Driven Architecture (EDA) is an architectural style in which one or more components in a software system execute in response to receiving one or more notifications. Event based programming differs from traditional web synchronous request-response interactions, since the main concepts are the events. Then, instead of speaking of clients and services, we refer to event producers and consumers. One of the main advantages of this architecture is that event producers and consumers can be decoupled, which improves its scalability and fault-tolerance capabilities. There are three main interaction styles in event programming [14]:

- *Push event distribution:* event producers emit an event and usually do not expect any specific action by event producers
- *Channel event distributions:* event producers send events to an event channel which acts as a broker, redirecting the event to event consumers subscribed to that particular event. This model is usually implemented using Message-oriented Middleware (MOM).
- *Pull event distribution:* event consumers follow the traditional request-response pattern to request an event from an event producers or from an event channel.

Event-based programming has been traditionally popular for programming user interfaces (e.g., Swing or JavaScript) as well as for integration architectures based on a Enterprise Service Bus. Given the requirements of the Live Web, event-based programming has given a step forward and is one of the cornerstones of highly interactive applications. We review in the following subsections *Node.js*, one of the most popular server-side programming environments, which is an example of the event oriented paradigm. Node.js applications are written in JavaScript and thus rely heavily on events.

III. MAIA architecture

An agent that does not interact with its environment (other software components, sensors, actuators, etc.) is of little practical use. For that reason, it is common practice to modify or extend agent platforms to include external sources. However, as previously explained, agent architectures tend to be monolithic. Connecting to external components is often a tedious and ad-hoc process. Regardless of the specific implementation, the resulting modifications are very heterogeneous and bound to the agent platform they were made for.

In an attempt to adapt generic BDI multi agent systems to seamlessly interact with different sources, we propose a new architecture, called MAIA.

The architecture has been designed to allow easy hot-plugging of new components that expand the capabilities of the system (e.g. new sensors). It consists of independent modules that perform different tasks (e.g. BDI reasoning, User Interface), which are connected using a common interface to a core platform that controls the flow of information between them.

Figure 1 shows an overview of the main modules in the architecture. At its core there is a bus for the modules that are closely related to a typical agent (BDI platform, sensors, actuators, etc.), another bus for the modules that connect to the Evented Web, and a central piece that connects both buses and provides additional services.

This section briefly presents these modules, focusing on the relationship between them. The following sections will describe each module separately in greater detail. The underlying communication mechanism is covered in Section IV.

First of all, the architecture includes a BDI Platform module which encapsulates all BDI functions and logic. This platform can be used to develop and run BDI agents that will communicate with the rest of the modules in the architecture.

An Adapter (labelled BDI Adapter) makes this communication possible by interfacing between the agent platform and the rest of the modules. Part of this adaptation is translating MAIA events to a format the platform understands, and vice versa. It will also make all the high level services from the rest of the modules available to the agents within the platform.

We use Jason as the reference BDI Platform in this paper, but any other platform such as Jade or Jadex would be suitable. The design of the BDI Adapter depends on the platform chosen.

The BDI Adapter is directly connected to the Agent Bus. The role of this bus is to connect the different high level modules of the agent, in contrast with the connectors to web services and other sources, which connect to the Evented Web Bus. This separation serves two main purposes: protecting the agent modules from an overload of events from the web, and providing additional capabilities to the modules connected to the Agent Bus (see Section III-B).

The Event Manager mediates between both buses, providing extra services to the Agent Bus as described in Section III-C. These services will have an important role in the development of BDI agents. Section III-A2 contains several plans and goals in Agent Speak that make use of these services.

A. Adapters

To be able to connect to any of the MAIA buses a module must communicate via events that are MAIA compliant (see Section IV) and use one of the protocols that its bus implements. Unfortunately, not all systems are natively evented. Even when they are, they do not always follow the MAIA events format or use the same protocol as the bus.

An Adapter is a piece of software that mediates between such systems and the rest of the modules. In the best case scenario, which is that of software that is already event oriented, the adaptation process is as simple as translating event formats on the fly and dealing with protocol differences.

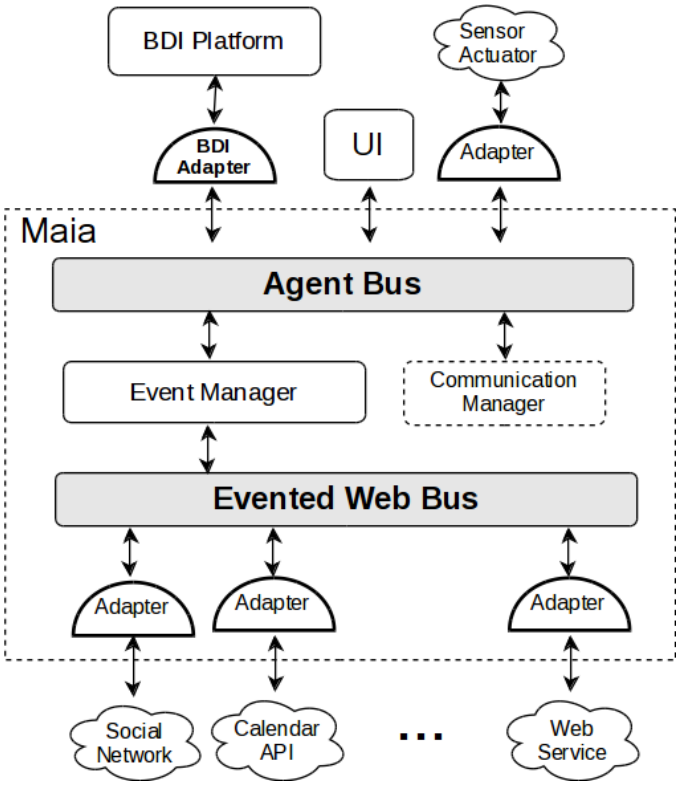

Fig. 1. High level representation of the MAIA architecture.

In the worst case scenario, deeper changes in the software itself might be needed.

We group the adapters in two categories according to the level of integration they provide: basic adapters and Agent Adapters. Basic adapters make the features of an external service or module available to the rest of the modules. Agent Adapters also make the advanced services provided by the Event Manager available to the module in question.

In essence, basic adapters simply add sources of information or interaction with external services, whereas an Agent Adapter connects to a module with more complex logic.

1) Basic Adapters: These adapters take care of: connecting with the Event Manager; translating event formats, back and forth; generating MAIA events and storing events for later consumption. Every adapter that connects to the Evented Web Bus is a basic adapter.

2) Agent Adapter: Agent Adapters are the interface between an agent system, typically an Agent Platform, and the Agent Bus. The role of these agent systems is to implement the logic of the final application, adding intelligence to the system and communicating to the different modules. The Event Manager provides several services to make it easier to perform certain common actions or simply delegate tasks that would otherwise be done by the agent. Thus, an Agent Adapter should integrate these services in the agent platform.

The design and features of the Agent Adapter highly depend on the target Agent Platform, its internals and the programming interface it offers. Hence, we will focus on the development of an adapter for Jason. Nevertheless, most of the concepts herein are general and can be used in other Agent Platforms.

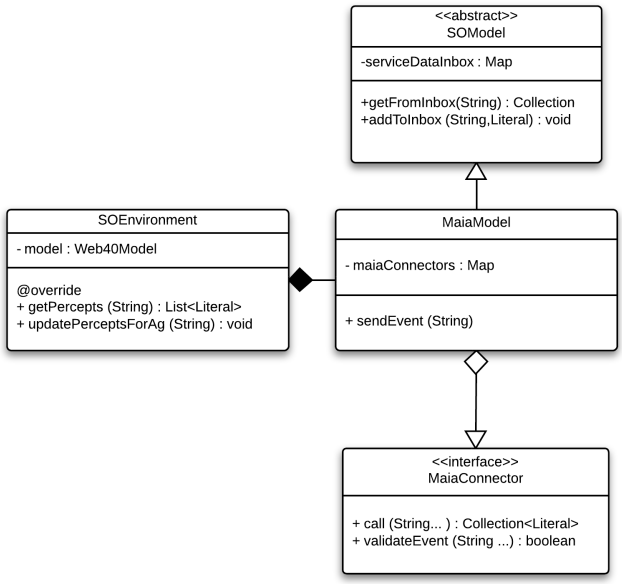

Fig. 2. Adding perceptions to agents in Jason

We identified three main challenges in the adaptation process. The first one consisted in communicating with the platform itself, and its individual agents. The second one was translating MAIA events to Jason beliefs. Lastly, there needs to be a way to use the extra services provided by the Event Manager from within any Jason agent. This section covers the first two, whereas Section V contains excerpts of Agent Speak code to deal with the most common MAIA services.

Every agent within Jason has its own knowledge database, which is populated by data from the different sources. To be able to actually modify the perceptions of the agents, a custom Jason Environment is needed, along with an ad-hoc model for this scenario. By modifying the basic Jason Environment we are able to control not only the sources through which new information is added, but the life cycle of such information.

More precisely, the custom model follows the data inbox concept, the same as regular mailboxes. All information received by the agent is volatile, and will be discarded after it is fetched. Should the agent find the information interesting or necessary for the future, it will save it as beliefs in its permanent knowledge database.

Using these data boxes it is rather easy to integrate our Java code and our agents in AgentSpeak. A special function allows any Java method to send information to any certain agent, and any Java function can be wrapped and made available to the agents in the platform. Figure 2 shows the custom elements created for the adapter.

Apart from the modifications explained above, events themselves need to be converted to beliefs internally. For this purpose, we created the libraries to translate a a subset of the JSON notation to beliefs and vice versa. Unfortunately,

the limited syntax of beliefs makes it impossible to perform a complete mapping.

Lastly, it is important to note that every agent should subscribe only to those events that are relevant to its functioning, and to avoid permanently storing them. Otherwise, we risk overloading the agents with too many facts, which hinders the reasoning process and might lead to undesired behaviours.

B. The Agent Bus and the Evented Web Bus

The role of the Evented Web Bus is to gather information from different web services and other non-web sensors, and to send information to those services when needed.

In addition to plain message passing, the bus has the following features: event filtering, event subscription and store and forward. Event Filtering provides the ability to select only the relevant events in each situation and for each module. By using Event Subscription modules can indicate their interest in certain kind of event which they wish to receive. Store and Forward means that modules can receive the events they subscribed to and that were sent while they were disconnected. It also means that events will be saved until they can be forwarded to a module. Without it, an overloaded module would not be able to consume all the events sent to it, which might then be discarded.

The Agent Bus connects the different modules that are directly related to the agents. The Agent Platform, the User-Interface and the Communication Manager are the most important examples of such modules.

In essence the Agent Bus works similarly to the Evented Web Bus. However, the modules connected are in charge of some of the highest level functions of the agent architecture. Thence they require some capabilities from the bus that were not necessary for the evented web. These capabilities are exposed to the agents in the form of services that highly ease the development of systems that take advantage of web services. Most of these services are focused on the development of personal agents that interact with social networks.

These services will be transparently provided to the modules in the Agent Bus by the Event Manager, covered in the following section.

It is important to note that the existence of these buses makes it possible to spread the modules that connect to it into several machines. Nonetheless, a simpler local configuration is possible.

C. Event Manager

The Event Manager is the core of the MAIA architecture. It is the bridge between the two buses. One of its roles is to exchange events between them, making Evented Web and sensory information available to agents and forwarding requests from agents to services. However, such information is usually verbose and frequent. Most of the times it is redundant or not critical. In contrast, the communication among agents or between agents and the user interface are usually more critical and sensitive to delays. As a consequence, the exchange between both buses obeys specific rules within the Event Manager. Such rules make the existence of two buses transparent to the clients of both while avoiding unnecessary forwarding between them.

Besides controlling the flow of events between the different modules, it complements the Agent Bus by providing higher level functions that are not present in it. The Event Manager provides several useful services for the development of personal agents.

Namely, these services are: Identity, Event Based Task Automation, Location, Semantic Information, Social Networks, Calendar and Transactions.

The Identity Service allows agents to define virtual identities. These identities can be linked to the rest of the services. For instance, an identity can be linked to several calendars and social networks. These identities are defined via FOAF [15]. Each identity has a unique ID that can be used to subscribe to the events from the sources linked to it. The Event Based Task Automation offers the option for agents to delegate actions to the Event Manager. These actions will be fired by a certain event, and their result will be another event.

The Social Network service homogenises the connection and interaction with different social networks. Social networks are an important part of the average user's everyday activity. By integrating them in a personal agent, we can gather relevant information about the user and improve the user's experience. Each social network profile can be linked to several identities. As we saw before, this means the events from different profiles will share a common namespace, making it easy to subscribe to all of them.

The Location service makes it possible to set locations to each identity. Events are sent every time there is a location change, or when a module queries the location of an identity.

The Calendar Service is a common interface to deal with calendars from different sources within Maia. It is especially meant as an abstraction for online calendar services.

The Information Service offers a simple unified interface for agents to query information from external information sources. As of this writing, the Information service supports SPARQL, being able to send queries to multiple endpoints (DBpedia, data.gov, etc.).

The Transaction service makes it easier for agents to handle operations with online services that follow a known pattern. For instance, the processes between booking a flight and arriving safe to the destination accommodation are quite similar regardless of the flight company, shuttle bus operator, etc. Given that, the Transaction service identifies different events as steps in such processes and acts accordingly to offer extra information to the agents.

IV. MAIA Events

The communication paradigm in MAIA purposely mimics that of the evented web [13]: all modules communicate through atomic messages called events. This paradigm follows the channel event distribution style.

The communication based on events is what confers loose coupling to the architecture. However, it also means that the structure and format of these events must cover a wide range

of scenarios. Furthermore, it is desirable to make events as compatible with the evented web as possible so that the interaction is seamless. This compatibility that must be achieved both in a conceptual level and in the format level.

The conceptual level deals with questions such as: what type of information does an event carry?, how do events relate to each other?, how are modules/services and events related? Most of these questions have already been answered in the previous sections, especially those related to the purpose and usage of events. The Live Web [13] introduces a very generic schema for events. However, a formal definition of the information within events is still missing.

The Evented Web Ontology (EWE) by Coronado et al. [16] formalises the idea of events on the web in the form of an ontology. The ontology itself was created after studying several task automation portals such as IFTTT. These portals either actively access services (web requests) or receive notification from them (web hooks). Either way, any new information from a services is modelled as an event. Users can choose what actions should be triggered when an event is detected (e.g. upload a picture to an image hosting site whenever there is new email with attachments). Interestingly, this scenario can be seen as a particular case of the evented web. The EWE vocabulary allows for such generalisation, which turns it into a consistent semantic model for representation of events. Hence, it provides the formal definition necessary for conceptual compatibility.

Describing EWE in depth is out of the scope of this paper. However, we will describe the concepts that are necessary to understand its use in this work and how it had to be expanded. Among other things, EWE defines Channels, Events and Actions. A Channel is a source of information, such as an e-mail inbox. Channels generate Events whenever there is new information, like whenever there is new mail. Each Channel also has a list of available Actions, like deleting an email.

In MAIA every new module is a Channel. For adapters, this Channel actually represents the source they are adapting. Additionally, an event can be either informative or a request, in the sense that it may inform of an action performed or of an intention to trigger an action in a remote entity. In other words, a module emits an event when there is new information to share, or when it expects another module to perform an action.

On the other hand, there are several possible formats to serialise semantic information. To simplify the task of developing new adapters to the evented web, MAIA events use the JSON-LD [17] format in its compact form. This approach has multiple advantages: it is a lightweight human-readable format; there are libraries to efficiently process JSON in almost every programming language and JSON-LD libraries have been made for most of them; semantic and non-semantic information can coexist in the same JSON object; and plain JSON information from the evented web might be converted to semantic JSON-LD by adding an appropriate context.

In summary, MAIA events are messages in JSON-LD format that are modelled using the EWE ontology. Events have the following fields:

- **id (@id)** Unique identifier of the sent event for the specified entity (source).
- **timestamp (dcterms:created)** Time of the original emission. This makes time reasoning possible and prevents the side effects of asynchronous communications.
- **source (ewe:source)** Unique identifier of the sending entity.
- **name (dcterms:title)** Which describes the event, and is the only required field. Ideally, it will not only consist of a basic string, but of a complete namespace. This allows for a complex processing of the events and an advanced filtering for triggers. We will get into details later in this section.
- **parameters (ewe:hasParameter)** For any kind of non-trivial event, we will need more information about the entities involved in the event, or the parameters if it is a request. This field is a list of ewe:Parameter objects, with description, title and value.
- **expiration** Used to announce other entities that after this time the success or error callbacks will not be called, to prevent them from replying to or acknowledging the event.

```
{
 "@context": {
  "ewe": "http://www.gsi.dit.upm.es/ontologies
      /ewe/ns",
  "dcterms": "http://purl.org/dc/terms",
  "id": "@id",
  "@type": "ewe:Event",
  "source": "ewe:source",
  "timestamp": {
   "@id": "dcterms:created",
  },
  "name": "dcterms:title",
  "parameters": {
    "@id": "ewe:hasParameter",
    "@container": "@list",
    "@type": "ewe:Parameter"
  },
  "description": "dcterms:description",
  "title": "dcterms:title",
  "value": "dcterms:value",
 },
 "id": "http://demos.gsi.dit.upm.es/maia#
     MailChannel_"
 "source": "http://demos.gsi.dit.upm.es/maia#
     MailChannel_ev_1389937684001
 "timestamp": 1389937684,
 "name": "MailChannel::email::new",
 "parameters": [
   {
     "title": "subject",
     "value": "Testing Maia",
     "description": "Subject of the email"
   }
 ],
 "expiration": 1389937694
}
```

Listing 1. Example of an event in MAIA that represents a MailChannel.

In addition to these fields, the complete JSON-LD object also includes a context to provide the semantic metadata of

each field. A complete example of an event can be seen in Listing 1

All events are named following a simple convention, the names are strings separated by double colons, the first string being the name of the module that sent it, for example: *MailChannel::email::new*. Modules use these names to subscribe to events from other sources. For instance, in our previous example a module would need to subscribe to *MailChannel::email::new* to receive the new email events from MailChannel.

What is interesting about MAIA events is that they may contain wildcards $\star$ or double wildcards $\star\star$. Using wildcards, a module can subscribe to a wide range of events. If the name of the event and the name used in the subscription match, the event will be forwarded. A single wildcard replaces/matches any string between double colons (e.g. *a::b::c* and *a::*::c* match). A double wildcard replaces/matches zero or more slots (e.g. *a::b::c* and ***::c* match, and also *a::b::c::***). Wildcards can be used either in the subscription name or in the event name, the comparison is applied symmetrically.

In order to efficiently process these matches and allow a high throughput of events, MAIA buses use an optimised subscription handling algorithm based on subscription trees.

Although one of the aims of the events system is to achieve asynchronous, it is worth noting that namespaces and the expiration information allow some sort of remote method invocation. To reply to an event, another event with the name `<source>::success::<id>` or `<source>::error::<id>` can be sent before *Expiration*, where *¡source¿* is the identifier of the sender and *¡id¿* is the ID of the original event. These events are currently not being forwarded to the rest of the modules.

As a last comment about the format of events, we have developed adapters for SPARQL and Spotlight endpoints. A W3C recommendation [18] can be used to include the results from SPARQL queries in events.

V. Case study: building a personal agent

To clarify some of the concepts explained before and put them in context, we will go through an example implementation of a personal agent in the travelling domain.

The aim of this personal agent is to assist users with their trips. This assistance includes: following the process between booking a ticket and arriving to the destination, alerting of any irregularity such as delays, cancellations or forecast alerts; informing users about flight deals during their free days; checking the activity on social networks about topics related to the trip; and handling emails and social activity on behalf of the users when they are away.

For all this to work, the agent will need to connect to: a flight search service; a forecast service; an email server; and a social network. The interaction between the user and the personal agent will be via text messages. The natural language processing of the messages from the user to an external REST Natural Language Understanding (NLU) Service. Each of the external services has an associated adapter module, as seen in Figure 3.

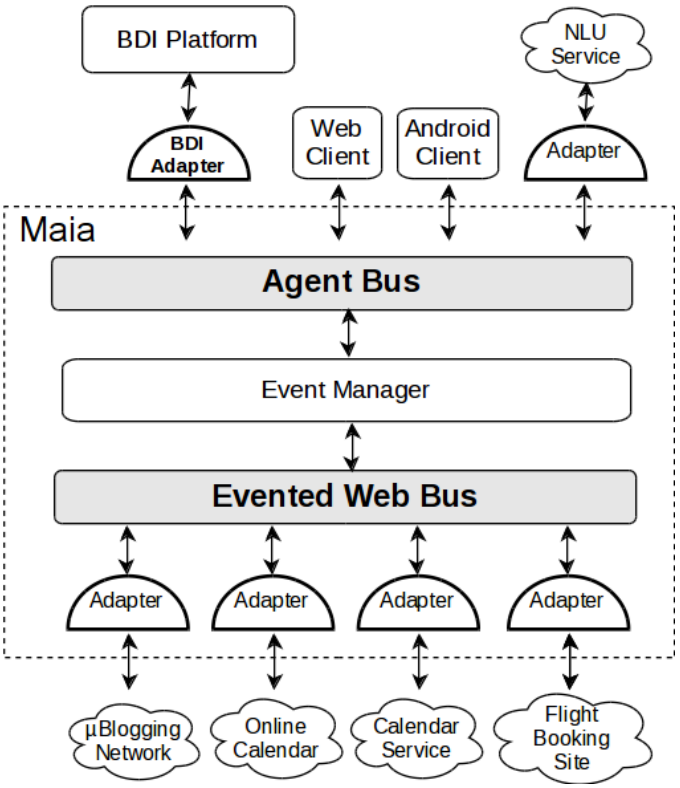

Fig. 3. Architecture of the Prototype.

The logic of the personal agent is provided by a single Jason agent, the travel agent. This section shows excerpts of code and simplified examples that demonstrate how to interact with the Event Manager to make use of its services. More specifically, it contains AgentSpeak plans to: get the semantic information of the country of the flight destination, which can later be used to fetch more information; alert the user via email when the user has confirmed a flight and the forecast information in the city of origin or destination is negative; subscribe to activity in all the subscribed microblogging sites about the country or city of destination two weeks before the flight, and alert the user about suspicious activity.

It is possible to simplify the syntax to emit frequent events, as seen in Listing 2

```
1 email(To,From,Subject,Body) :- parameters((
      name("to"),value(To)),(name("from"),value
      (From)),(name("subject"),value(Subject))
      ,(name("body"),value("body")))).
2 sendEmail(To,From,Subject,Body) :- event(["
      action","email","send"],email(To,From,
      Subject,Body)).
```

Listing 2. Definitions for email handling.

Listing 3 contains a plan to process forecast information during or close to a day of a scheduled flight. To receive such forecast information, the agent must have already subscribed to forecast alerts or any event from the information service.

Listing 4 exemplifies how an agent can query a SPARQL endpoint to get more information. In particular, it fetches the capitals of the capitals in Europe if a new flight is booked but

the country of the destination city is not known. The query is limited to European cities to use a simple query to a public endpoint (DBpedia).

```
+info("forecast",data(Date,City,Temperature,
    Forecast,Chances))
  : flight(Dept,City,From,To)[id(Identity)] |
      flight(City,Arriv,From,To)[id(Identity
    )]  ((Temperature < 20 | Forecast ==
    "rain" )  Chances > 0.3 )
  <-!suggest_deals(Identity,Dept,Arriv,From,
      To);
    sendEmail(email_address(Identity),null,"
        Bad weather for your trip",(Date,
        Temperature,Meteo,Chances)).
```

Listing 3. Process forecast information when a flight has been scheduled.

```
+flight(_,City,_,_)
  : ~country(City,_)
  <-query_sparql("
    SELECT distinct ?country ?capital (SAMPLE
        (?caplat) AS ?caplat) (SAMPLE(?
        caplong) AS ?caplong)
    WHERE {
     ?country rdf:type dbpedia-owl:Country .
     ?country dcterms:subject <http://dbpedia
         .org/resource/Category:
         Countries_in_Europe> .
     ?country  dbpedia-owl:capital ?capital .
     OPTIONAL {
       ?capital geo:lat ?caplat ;
         geo:long ?caplong .
     }
    }
    ORDER BY ?country
    ",country(1,2),location(2,3,4,_)).
```

Listing 4. Demonstrates how to use a SPARQL query to gather new information.

Lastly, Listing 5 presents a simple example which makes use of the social service. More specifically, the agent subscribes to microblogging events up to fifteen days before a flight is scheduled to depart. The social service will then send alerts about activity when there are enough microblogging posts related to the destination city or country. It is easy to imagine that this feature is helpful to detect noteworthy happenings in the destination country (riots, strikes, concerts, etc.)

```
+flight(_,City,Dept(YY,MM,DD,_,_,_),_)[id(
    UserID)]:
  : ((DD > 15 .date(YY,MM,DD-15)) |
      (.date(YY,MM-1,DD+15)))
       country(City,Country)
  <-social(event(["id",UserID,"social","
      ublogging","**","stream","peak"), [
      Country,City], ["alert","activity","
      ublogging","away"]).

+event(["alert", "activity", "ublogging", _],
     data(Volume, Posts))[id(Identity)]
  : Volume > 10
  <-ui_alert(Identity, "Relevant info from
      the social networks about your
      destination:", Posts).
```

Listing 5. Subscribe to notifications about peaks in activity about the destination of a trip and warn the user via the UI upon alert.

The interaction with the user can be done via a Web client (a Google Chrome extension that connects to the Agent Bus), or an Android application. Both clients also send the location of the user, so they are both UIs and sensors.

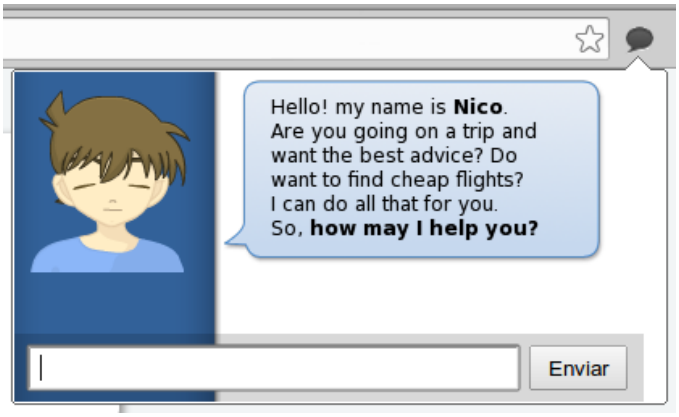

Fig. 4. User Interface as a Chrome Extension

VI. Related work

Several authors have addressed the definition of an event based agent architecture. Munteanu [19] proposes an event-based middleware for Cloud Governance based on multiagent system. Their work is focused on identifying the agent roles for cloud governance and does not deal with engineering an event-based agent system. Thus, our solution can complement their proposal since it provides a suitable architecture for event-based processing.

In the first prototypes of this system, different multi agent system platforms were evaluated. The most promising of them being SPADE (Smart Python multi-Agent Development Environment) [20], as it includes the XMPP protocol in its core and many of its communication features and its advantages: publish-subscribe mechanism to allow push updates, form-data to manage work-flow between user, libraries for many programming languages and platforms, etc..

So far, we referred to communication between modules in the general sense. The elements mentioned make it possible to exchange information between different parties. However, agent communication is a more sophisticated process that has been treated broadly in other texts [10], which describe complex agent communication solutions. Although MAIA focuses on a different problem, it was designed so that these solutions are compatible with and can be implemented on top of it. To make this possible, two possible additions might be needed: one in the agent level, adding the communication logic and protocols; and another one on the platform level, which allows agents to announce or subscribe their services, share protocol definitions or that acts as a mediator in disputes. The first addition would be made on top or within the MAIA adapter, if it is not already contemplated in the agent platform. The second one is labelled as Communication Manager module in the MAIA architecture. This paper will not cover this specific module, but it is important to note that the architecture was created with it in mind.

VII. Conclusions and future work

The architecture presented in this paper proves that it is possible to achieve modern systems that combine the potential of intelligent agent systems and the interconnection and ever-growing applications of the modern web.

The resulting application goes beyond the state of the art, putting together already existing solutions from different fields. It thus shows that we can make good use of the existing technologies to implement innovative ideas.

It is important to note that the most important shift is in the way we understand agents and agent communication. Adapting existing systems and frameworks to MAIA also requires work, especially in the case of Multi Agent Systems. However, such adaptation only needs to be done once, and it allows its connection to a wide range of modules.

There are several aspects in which MAIA can be extended or improved. It also opens the discussion about the integration of the evented programming paradigm and the design of BDI agents.

One of the main aspects to improve from a pragmatic point of view is the security of the information being exchanged and the scope in which it is visible. Currently MAIA allows username/password authentication and mechanisms to control event subscription on a per-module basis.

Another field for future research is to further expand the definition of events to include other concepts such as propagation of events. This might lead to delegation and collective planning, but it also poses challenges related to agent communication.

References

[1] J. P. Müller and M. Pischel, "The agent architecture interrap: Concept and application," German Research Center for Artificial Intelligence (DFKI), Tech. Rep., 1993.

[2] J. P. Müller, "Architectures and applications of intelligent agents: A survey," *The Knowledge Engineering Review*, vol. 13, no. 4, pp. 353–380, 1999.

[3] A. Pokahr and L. Braubach, "From a research to an industry-strength agent platform: Jadex v2," *Business Services: Konzepte, Technologien, Anwendungen. 9. Internationale Tagung Wirtschaftsinformatik*, pp. 769–780, 2009.

[4] P. Wallis, R. Ronnquist, D. Jarvis, and A. Lucas, "The automated wingman - using jack intelligent agents for unmanned autonomous vehicles," in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 5, pp. 5–2615–5–2622 vol.5.

[5] R. H. Bordini and J. F. Hübner, "Bdi agent programming in agentspeak using jason," in *Proceedings of 6th International Workshop on Computational Logic in Multi-Agent Systems. Volume 3900 of lncs*. Springer, 2005, pp. 143–164.

[6] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[7] "Foundations for intelligent physical agents (FIPA)," 2001, available from http://www.fipa.org.

[8] A. S. Rao, M. P. Georgeff *et al.*, "Bdi agents: From theory to practice," in *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, 1995, pp. 312–319.

[9] I. Bratman, "Plans, and practical reason," *Cambridge, Mass.: Harvard UP*, 1987.

[10] D. Lillis, "Internalising Interaction Protocols as First-Class Programming Elements in Multi Agent Systems," Ph.D. dissertation, University College Dublin, 2012.

[11] B. Alfonso, E. Vivancos, V. Botti, and A. García-Fornes, "Integrating jason in a multi-agent platform with support for interaction protocols." in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11 and VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011, pp. 221–226. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095084

[12] D. Greenwood, M. Lyell, A. Mallya, and H. Suguri, "The ieee fipa approach to integrating software agents and web services," in *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ser. AAMAS '07. New York, NY, USA: ACM, 2007, pp. 276:1–276:7. [Online]. Available: http://doi.acm.org/10.1145/1329125.1329458

[13] P. Windley, *The Live Web: Building Event-Based Connections in the Cloud*. Course Technology, 2011. [Online]. Available: http://books.google.es/books?id=_AxfXwAACAAJ

[14] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Co., 2010.

[15] D. Brickley and L. Miller. (2014, Jan.) Foaf vocabulary specification. [Online]. Available: http://xmlns.com/foaf/spec/

[16] M. Coronado and C. A. Iglesias. (2013) Ewe ontology: Modeling rules for automating the evented web. GSI. [Online]. Available: http://www.gsi.dit.upm.es/ontologies/ewe/

[17] M. S. et al. (2014, Jan.) Json-ld 1.0. [Online]. Available: http://json-ld.org/spec/latest/json-ld/

[18] A. Seaborne. (2011, Jan.) Sparql results in json. [Online]. Available: http://www.w3.org/TR/sparql11-results-json/

[19] V. I. Munteanu, T.-F. Fortis, and V. Negru, "An event driven multi-agent architecture for enabling cloud governance," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, ser. UCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 309–314. [Online]. Available: http://dx.doi.org/10.1109/UCC.2012.50

[20] M. E. Gregori, J. P. Cámara, and G. A. Bada, "A jabber-based multi-agent system platform," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, 2006, pp. 1282–1284.